Computer Science Department

TECHNICAL REPORT

ASSET User Manual

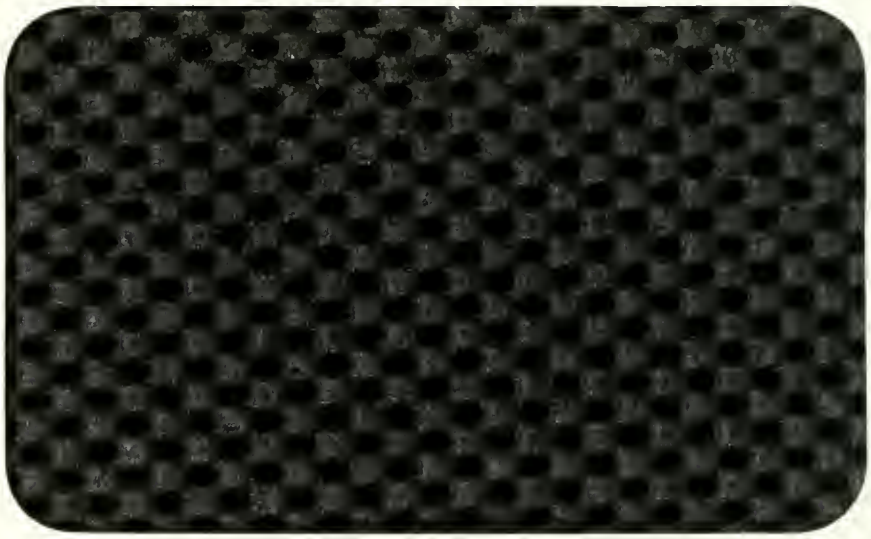
Phyllis Frankl

Technical Report 318

October 1987

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



ASSET User Manual

Phyllis Frankl

Technical Report 318

October 1987

ASSET USER MANUAL
October 1, 1987

Phyllis Frankl

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, New York 10012

1. INTRODUCTION

ASSET (A System to Select and Evaluate Tests) is an interactive software testing tool based on data flow testing [1,2,3,4]. It takes as input a Pascal program, a set of test data, and one of the data flow testing criteria. It runs the program on the test data and produces a list of those definition-use associations required by the given criterion but not covered by the test data.

The user then examines the output to determine whether the program has behaved according to its specification. If the program computed the wrong output, the user reports that the program has a bug. If the program has behaved correctly on all of the test cases, and if the criterion has not been satisfied, the user selects additional test data and continues the testing process. If the program has behaved correctly and the criterion has been satisfied the user can either release the program, certifying that it has been adequately tested according to the criterion, or can select another criterion and continue.

ASSET is used for testing an *individual subprogram* (procedure, function, or main program) which we'll call the "subject procedure". We will refer to the program of which the subject procedure is part as the "subject program". The data flow analysis which it performs is *intra-procedural* analysis of the subject procedure.

This manual assumes that the reader has some familiarity with the theory of data flow testing and with the UNIX operating system. To acquire some familiarity with data flow testing, see the references in section 7. Section 2 of this manual defines the restrictions on programs accepted by ASSET. Section 3 describes the most important ASSET commands. Section 4 describes the rest of the ASSET commands. Section 5 describes how to use ASSET on a separately compiled program. Section 6 gives an example of an ASSET session. Section 7 gives references to relevant papers. Appendix I describes how to install ASSET.

Acknowledgements:

The graphics facility was written by Stewart Weiss, who also helped implement an earlier version of ASSET. The user-interface was translated from csh to C and enhanced by Ernie Campbell.

Disclaimer:

ASSET is still a "work in progress". We do not vouch for its reliability nor are we responsible for maintaining it. However, if you find any bugs, or have other suggestions as to how the tool could be improved we would appreciate hearing from you. Please send e-mail to frankl@csd2.nyu.edu *and* to weyuker@csd2.nyu.edu.

2. THE LANGUAGE ACCEPTED BY ASSET

ASSET accepts *syntactically correct* programs written in the subset of (level 0) ISO Pascal consisting of programs which **do not** include any of the following constructs:

1. **goto** statements and labels.
2. functions with **var** parameters.
3. variant records
4. **with** statements
5. procedural or functional parameters
6. **forward** declarations
7. conformant arrays

The following identifiers are used in the modified version of the subject program and therefore should not be used in the subject program:

FW
traversed

The following identifiers are used as file names by ASSET and therefore should not be used as file names in the subject program:

IncFiles, StringTabFile, SymTabFile, anomalies, a.out, comp.msgs, copy, copy.exec, copy.p, copy2, copy2.p, defuse, display, globdecs, graph, junkfile, listing, mainfile, mainfile2, pairs, paths, picfile, record, results, results.old, subject, traversed

It is recommended that the user compile the program before submitting it to ASSET in order to insure that it is syntactically correct.

In addition, ASSET accepts certain Berkeley Pascal constructs which are not part of standard Pascal, including separate compilation (see section 5).

ASSET places certain arbitrary restrictions on the size of the program. Specifically,

number of variables visible to subject procedure <= 300
number of procedures and functions visible to subject procedure <= 300
number of types visible to subject procedure <= 300
number of constants visible to subject procedure <= 300
number of identifiers visible to subject procedure <= 500
length of each identifier <= 80
total number of characters in all identifiers visible to subject procedure <= 3000
number of basic blocks in subject procedure <= 100

For instructions on how to increase these limits, see Appendix I.

BUGS:

1. ASSET requires that the last statement in a case statement be followed by a semi-colon. For example, the (syntactically correct) program fragment

```
case b of
      true: writeln('hello');
      false: writeln('goodbye')
end
```

will be rejected by ASSET, but

```
case b of
```

```
                true: writeln('hello');
                false: writeln('goodbye');
        end
```

will be accepted.

The user should check that all case statements in the subject procedure are of this form *before* submitting the program to ASSET.

2. ASSET translates **for** loops incorrectly: If a statement of the form

"**for** i := e1 **to** e2 **do** S"

appears in the subject procedure, in writing the modified subject procedure ASSET will translate this to a code fragment functionally equivalent to

```
     i := e1;
<label1>:    if i > e2 then goto <label2>
          S;
          i := succ(i);
          goto <label1>;
<label2>:    next statement;
```

It should translate it to a fragment functionally equivalent to

```
     i := e1;
     tmp := e2;
<label1>:  if i > tmp then goto <label2>
           S;
           i := succ(i);
           goto <label1>;
<label2>:    next statement;
```

Thus, if any variable occurring in expression is modified within the statement S (which is legal, though poor programming practice), the modified subject program will not be equivalent to the subject program.

These bugs will be corrected in the next version of ASSET.

3. THE MOST IMPORTANT ASSET COMMANDS

In this section we describe the basic ASSET commands. The commands are listed in an order in which they can be executed. The section ends with some comments on the relationship between the commands. Additional commands are described in the next section.

Invoking ASSET

We will refer to the directory from which ASSET is invoked as the "current working directory". Before invoking ASSET you should set up a "subject directory" which contains the source of the program to be tested. It is usually convenient to have the subject directory be a subdirectory of current working directory, but this is not necessary. The subject directory *should not* be the same as the current working directory. To enter ASSET type "asset" in response to the UNIX prompt. ASSET will respond with

```
        Welcome to ASSET. For help type "help."


        Enter relative pathname of initial default directory. >>:
```

At this point you should enter the pathname of the subject directory, relative to the current working directory. ASSET will then print the ASSET "command level prompt",

">>>: ", signifying that it is ready to accept a command.

We now describe each of the ASSET commands. These commands can be invoked when ASSET types the command level prompt. Some of these commands will query the user for additional information. When this occurs, ASSET will prompt the user with its "inner prompt", ">>: ". Sometimes ASSET will ask the user a question which has a yes or no answer. The user should respond with "Y" or "y" for yes and "N" or "n" for no. ASSET will indicate the default (if there is one) in brackets. If the default is "Y" then any non-null response which does not begin with "N" or "n" will be considered to be "Y". Similarly, if the default is "N" then any non-null response which does not begin with "Y" or "y" will be considered to be "N". If the user just hits carriage return, the inner prompt will continue to appear until a non-null response is given. The logical dependence of ASSET commands is given in Figure 1. The commands must be typed in lower case letters. Any command can be abbreviated by its first three letters.

begin

The "begin" command analyzes the subject procedure, breaking it into basic blocks, constructing its flow graph, and determining the blocks in which each variable has definitions and c-uses and the edges on which each variable has p-uses. An adjacency list for the graph is written to the file "graph", and a representation of the data-flow information is written to the file "defuse". The "String Table" in which ASSET has recorded the identifiers appearing in the subject program is written to the file "StringTabFile".

In addition, ASSET constructs a modified version of the subject program in which probes have been inserted. That is, at the beginning of basic block i, the statement "writeln(traversed,i :FW);" is inserted. Declarations of the text file "traversed", the constant "FW", and any labels which appear in the modified program are also inserted. We will refer to the resulting program as the "modified subject program". ASSET writes a pretty-printed version of the modified subject program to the file "copy.p".

ASSET also creates files "copy", "copy2" (preliminary versions of the modified subject program) and "SymTabFile" (a representation of the Symbol table for the subject program) which will not be used by subsequent commands, but may be useful for debugging.

When you type "begin", ASSET will respond with

Enter name of subject procedure file. >>:

If there is no file of this name in the subject directory ASSET will print an error message and return to the command level. Otherwise ASSET will copy the file to the file "subject", then say

Separate Compilation? (Y/N)[N] >>:

Answer "y" if the subject program is separately compiled, "n" otherwise. We will assume throughout the rest of this section that the subject program is NOT separately compiled. See section 5 for instructions on using ASSET on separately compiled programs.

Next, ASSET will ask you for the name of the procedure to be instrumented for testing. You may either supply the name of the procedure, or direct ASSET to prompt you with the names of the procedures in the program.

select

The "select" command causes ASSET to print a menu and query the user to select one of the data flow testing criteria.

find-associations

This command causes ASSET to produce a list of all definition-use associations (of the appropriate type) in the subject procedure. If the selected criterion is All-du-paths, find-associations writes a list of all of the du-paths in the subject procedure to the file "paths". Otherwise, find-associations writes a list of all of the definition-c-use associations and definition-p-use associations in the subject procedure to the file "pairs". Note that in the worst case, finding all of the du-paths may take time exponential in the number of basic blocks.

compile

The "compile" command invokes the Pascal compiler to compile the modified subject program. The executable code is written to the file "copy.exec". Warning and Error messages generated by the compiler are written to the file "comp.msgs".

run

The "run" command executes the modified subject program, on one or more test cases and adds the program trace to the file "traversed". When you type "run", ASSET may respond with

> File 'traversed' already exists. Type 'Y' if you want to append to it, 'N' otherwise. >>:

Type "Y" if you want to add the program traces generated by previous test cases to the current one, "N" otherwise. If you answer no, ASSET will ask you whether you want to save the old file "traversed" (i.e. the program traces of previous test cases).

Next, ASSET will ask you whether the program takes input from command line arguments. Answer yes if the subject program reads command line arguments via the Berkeley Pascal argv mechanism or if you would like to redirect the standard input or output using the UNIX redirection mechanism. If you answer yes, ASSET will prompt you for the command line arguments. For example supplying

> <infile >outfile

as the command line argument will cause the modified subject program to read its standard input from the file "infile" and write its standard output to the file "outfile".

ASSET will then say

> Executing modified subject program ...

then will execute the modified subject program with the command line arguments you've supplied. When execution terminates, ASSET will ask you whether you would like to run the program on additional test data. If you answer yes, ASSET will again ask you for command line arguments and execute the program. The new program trace will be added to the traces which have previously been written to the file "traversed". If you answer no, ASSET will return to the command level.

Note that one trace is produced for each run of the *subject program*, not one for each invocation of the *subject procedure*. Thus a single program trace may contain several paths from the entry to the exit of the subject procedure. If the subject procedure is recursive these paths will be "nested" within one another.

check

The "check" command checks whether the selected criterion is satisfied by the test data whose program traces are in the file "traversed". ASSET writes to the file "results" a list of those def-use associations which are required by the criterion but have not been

covered by the test data. (Before doing this, ASSET moves the old version of the file "results" to the file "results.old". This allows the user to easily see which def-use associations were eliminated by the most recent additions to the test set by using the UNIX command "diff results results.old"). If the file "traversed" does not exist, ASSET initializes it to represent an empty set of program traces. In this case, "results" will contain a list of all of the def-use associations required by the criterion.

save

The "save" command moves the files "subject", "copy.p", "copy.exec", "traversed", "results", "pairs", "paths", "defuse", etc. from the current working directory to the subject directory.

exit

Moves important files to the subject directory and removes the rest, then exits.

Note on the relationship between the commands

Before checking whether a set of test data satisfies any of the criteria All-defs, All-p-uses, All-c-uses, All-c-uses/some-p-uses, All-p-uses/some-c-uses or All-uses it is necessary for ASSET to create a list of definition-use associations in the file "pairs". Before checking whether a set of test data satisfies the criterion All-du-paths it is necessary for ASSET to create a list of du-paths in the file "paths". When the most recently selected criterion is All-defs, All-p-uses, All-c-uses, All-c-uses/some-p-uses, All-p-uses/some-c-uses or All-uses the "find-associations" command creates the file "pairs". When the most recently selected criterion is All-du-paths the "find-associations" command creates the file "paths". Thus it is necessary to use the "find-associations" command *once* before the first time that "check" is invoked with one of the criteria involving "pairs" and *once* before the first time that "check" is invoked with the criterion du-paths.

Suppose the user has been attempting to satisfy criterion C then decides to switch to another criterion, C'. Let T be the set of test data on which the program has just been run (i.e. assume that set of program traces in the file "traversed" corresponds to test set T). Then to see how close T comes to satisfying C' the user need only invoke the "select-criterion" command (to select the new criterion, C') and the "check" command. It is not necessary to run the program on test set T again.

4. OTHER ASSET COMMANDS

csh

Spawns a new shell from which you can invoke your favorite UNIX commands. To exit from this shell and return to the ASSET command level type control-D.

directory

Displays the name of the subject directory and prompts the user for a new subject directory.

graphics

The "graphics" command writes a PIC program describing a graphical representation of the subject procedure's flow graph. The PIC program is written to the file "picfile". To obtain a picture of the flow graph, run the troff preprocessor "pic" (from a

UNIX shell, not directly from ASSET) with picfile as input then pipe it through troff.

[At NYU, pipe the pic output through itroff (for output on a Canon Laser Printer), psroff (for output on an Apple Laser Writer), or preroff (for output on the console if you are in the Suntools environment.) For example, typing

```
% pic picfile | psroff -Pap3
```

(where "%" is the UNIX prompt) will send printed output to the LaserWriter in room 505.]

help

On line help facility.

purge

Removes (i.e. destroys) all of the files "subject", "copy.p", "copy.exec", "traversed", "results", "pairs", "paths", "defuse", etc. from the current working directory. Handle with care.

sepcomp

Compiles and links separately compiled subject program. See section 5.

restore

Copies the files "subject", "copy.p", "copy.exec", "traversed", "results", "pairs", "paths", "defuse", etc. from the subject directory to the current working directory, thus restoring a previous ASSET session. If the files in the subject directory were created by running ASSET on a separately compiled subject program some additional action is taken (see section 5.)

view

The "view" command directs ASSET to display the contents of a (text) file on the screen, a screenful at a time. If you type "view <filename>" ASSET will display that file. Otherwise, ASSET will prompt you for the file name. The view command works by invoking the UNIX command "more", hence responds to the same inputs as does "more". Most importantly, type carriage return to display the next line of text, space to display the next screenful of text and q to return to the ASSET command level. For other options, see the UNIX manual page for "more".

5. USING ASSET ON SEPARATELY COMPILED PROGRAMS

This section describes how to use ASSET to test separately compiled programs. This section assumes some familiarity with Berkeley Pascal's separate compilation mechanism. (See the relevent sections of the Berkeley Pascal manual.)

To instrument a procedure for testing, ASSET must create modified versions of the file containing the source for that procedure, the file containing the source for the main program, and a file containing global variable declarations. To create an executable code for the modified subject program, ASSET must recompile and relink the modified files and any other files dependent on them.

In order to do this, ASSET assumes that subject directory contains a Makefile describing the dependency information (see documentation for the UNIX command "make"), that there is a file called "globals.h" which has global variable declarations, and such that the object files for the main program and for the procedure to be instrumented

are dependent on globals.h.

If the user answers "yes" to the question "Separate compilation?" during execution of ASSET's "begin" command, ASSET prompts the user for the names of the file containing the main program and the file containing the subject procedure. We'll refer to these as <main_file> and <subject_proc_file>, respectively. ASSET moves <main_file>, <subject_proc_file>, and "globals.h" to <main_file>.bak, <subject_proc_file>.bak, and "globals.h.bak", respectively. ASSET writes the modified versions of the files to <main_file>, <subject_proc_file>, and "globals.h". Thus when the UNIX "make" command is invoked, the modified subject program will be compiled and linked.

To compile the modified subject program use the ASSET command "sepcomp", rather than the ASSET "compile" command. Sepcomp invokes the UNIX make command, and moves the executable code (which is assumed to be in "<subject-directory>/a.out") to copy.exec.

When the "save" command is invoked, ASSET looks for files with the ".bak" extension in the subject directory. For each such file, ASSET moves <fname> to <fname>.mod and moves <fname>.bak to <fname>. Thus the original files have the same contents as they had before the ASSET session, and the modified versions have the extension ".mod".

The "restore" command looks for files with the ".mod" extension and queries the user as to whether they should be restored. If the user answers yes to the question "restore <fname>.mod (Y/N)?" then ASSET moves <fname>to <fname>.bak and moves <fname>.mod to <fname>. Thus to restore a session in which the instrumented procedure was in file "Proc3.p", the main program was in file "main.p", answer yes to the questions "restore Proc3.p.mod?", "restore main.p.mod?", and "restore globals.h.mod?"

Restrictions on separately compiled programs :

The separate compilation option makes the following assumptions:

1. All relevant files are in the subject directory.
2. There is a file called "globals.h" containing global definitions in the subject directory.
3. "globals.h" is mentioned in #include directives in the subject procedure file and in the main program file.
4. There is a Makefile containing the dependency information, etc. needed to re-link and load the program contained in the subject directory.
5. The Makefile directs the executable code to file "a.out".
6. The procedure being instrumented is at the outer level of nesting.
7. The last included .h file cannot have any extra garbage at the end. (This is a bug).

NB -- It is currently NOT POSSIBLE to instrument the main program of a separately compiled module.

6. EXAMPLE OF AN ASSET SESSION

In this section we present an annotated example of an ASSET session. To distinguish between text written by the system and that written by the user, we display text entered by the user in boldface type. Comments are written in italics.

Example 1:

This example shows an ASSET session in which a brute-force string-matching procedure is analyzed. The program reads a string and a pattern. It is supposed to print the position in the string at which the pattern first appears, and print 0 if the pattern never appears in the string. The current working directory has a subdirectory called "StrMtch". The file "StrMtch/subject.p" contains the following program:

```
program TestStringMatch (input,output);
const MAX = 80;
      LENGTH = 10;
type  Source = array[1..MAX] of char;
      String = array[1..LENGTH] of char;
var   Pat : String;
      Txt : Source;
      i,result,TxtLen  : integer;

      function StringMatch(Pattern:String; SorText: Source;
            PatLen, SorLen : integer): integer;
      {Brute force pattern-matcher. Returns 0 for no match}
      var PatPos, SorPos : integer;
      begin
            PatPos := 1;
            SorPos := 1;
            repeat
                  if Pattern[PatPos] = SorText[SorPos] then
                  begin
                        SorPos := SorPos + 1;
                        PatPos := PatPos + 1
                  end  {then}
                  else
                  begin
                        SorPos := (SorPos - PatPos) + 2;
                        PatPos := 1
                  end; {else}
            until (PatPos > PatLen) or (SorPos > SorLen);
            if PatPos > PatLen
              then StringMatch := SorPos - PatLen
              else StringMatch := 0
      end;  {StringMatch}

begin {main program}
      {Read Input}
      writeln('ENTER THE TEXT');
      i := 1;
      while (not(eoln) and (i <= MAX)) do
      begin
            read(Txt[i]);
            i := i + 1
      end;
      TxtLen := i - 1;
      readln;
```

```
        writeln('ENTER THE PATTERN');
        i := 1;
        while (not(eoln) and (i <= LENGTH)) do
        begin
                read(Pat[i]);
                i := i + 1
        end;
        readln;

        result := StringMatch(Pat,Txt,i - 1, TxtLen);
        writeln('The pattern first appears at position ',result:3,
                        ' in the text.');
end.
```

Script started on Sun May 31 13:09:59 1987
csd27> **asset**

Welcome to ASSET. For help type "help."

Enter relative pathname of initial default directory.
>>: **StrMtch**

>>>: **begin**
Enter name of subject procedure file.
>>: **subject.p**
Separate Compilation? (Y/N) [N]
>>: **n**
Enter the name of the procedure to be instrumented.
If you would like to be prompted with the names of
the procedures in the subject program, just hit carriage return.
>>:

==> Should TestStringMatch
be instrumented for testing? (Y/N)
>>: **n**
==> Should StringMatch
be instrumented for testing? (Y/N)
>>: **y**

>>>: **select**

SELECT A CRITERION

A. All-defs
B. All-c-uses
C. All-p-uses
D. All-c-uses/some-p-uses
E. All-p-uses/some-c-uses
F. All-uses

G. All-du-paths

Enter letter representing the selected criterion
>>: **a**
Criterion is All-defs.

>>>: **find**

We next check whether the criterion has been satisfied with no test data. This is not necessary, but by doing this, we get a list of all of the def-c-use and def-p-use associations in the program.
>>>: **check**
ALL-DEFS:

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
Pattern	1	(3, 5)
Pattern	1	(3, 4)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorText	1	(3, 5)
SorText	1	(3, 4)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatLen	1	8
PatLen	1	(6, 3)
PatLen	1	(6, 7)
PatLen	1	(7, 9)
PatLen	1	(7, 8)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorLen	1	(6, 3)
SorLen	1	(6, 7)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatPos	2	4
PatPos	2	5
PatPos	2	(3, 5)
PatPos	2	(3, 4)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorPos	2	4
SorPos	2	5
SorPos	2	(3, 5)
SorPos	2	(3, 4)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatPos	4	4
PatPos	4	5
PatPos	4	(3, 5)
PatPos	4	(3, 4)
PatPos	4	(6, 3)
PatPos	4	(6, 7)
PatPos	4	(7, 9)
PatPos	4	(7, 8)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorPos	4	4
SorPos	4	5
SorPos	4	8
SorPos	4	(3, 5)
SorPos	4	(3, 4)
SorPos	4	(6, 3)
SorPos	4	(6, 7)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatPos	5	4
PatPos	5	5
PatPos	5	(3, 5)
PatPos	5	(3, 4)
PatPos	5	(6, 3)
PatPos	5	(6, 7)
PatPos	5	(7, 9)
PatPos	5	(7, 8)

AND
Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorPos	5	4
SorPos	5	5
SorPos	5	8
SorPos	5	(3, 5)
SorPos	5	(3, 4)
SorPos	5	(6, 3)

SorPos 5 (6, 7)

To look at these again use the command 'view results'.

Next we will compile the program and start running it on some test data. As the initial test data set, we select one element in which the pattern appears in the string and one element in which the pattern does not appear in the string.

>>>: **compile**
Compilation begins ...
Done, and successful.

>>>: **run**
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: **n**
Do you want to save old 'traversed'? (Y/N) [N]
>>: **n**
Command line arguments? (Y/N) [Y]
>>: **n**

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quick
The pattern first appears at position 5 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: **y**
Command line arguments? (Y/N) [Y]
>>: **n**

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quack
The pattern first appears at position 0 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: **n**

>>>: **check**
ALL-DEFS:
CRITERION SATISFIED

To look at these again use the command 'view results'.

The test set satisfies the all-defs criterion. We next check whether the same test set satisfies a stronger criterion, all-uses.
>>>: select

SELECT A CRITERION

A. All-defs
B. All-c-uses
C. All-p-uses
D. All-c-uses/some-p-uses
E. All-p-uses/some-c-uses
F. All-uses
G. All-du-paths

Enter letter representing the selected criterion
>>: f
Criterion is All-uses.

>>>: check
ALL-USES
Still need to exercise all of the following of def-clear paths:

with respect to	from	to
PatPos	2	4
SorPos	2	4
SorPos	5	8
PatPos	2	(3, 4)
SorPos	2	(3, 4)
PatPos	4	(7, 9)
PatPos	5	(7, 8)

To look at these again use the command 'view results'.

To aid in the selection of test data which cover the remaining def-use associations, the user can draw the flow graph (see Figure 2) and use "copy.p" to aid in labeling each node with the corresponding code. Notice that for each i, block i begins with the statement "write(traversed,i:FW);"

>>>: view copy.p

```
program TestStringMatch(traversed, input, output);

var
   traversed: text;
const
   MAX = 80;
   LENGTH = 10;
type
   Source = array [1..MAX] of char;
   String = array [1..LENGTH] of char;
var
```

```
Pat: String:
Txt: Source;
i, result, TxtLen: integer;

function StringMatch(Pattern: String; SorText: Source; PatLen, SorLen: integer): integer;

label
  10;
const
  FW = 4;
var
  PatPos, SorPos: integer;

begin

  write(traversed, 1: FW);
  write(traversed, 2: FW);
  PatPos := 1;
  SorPos := 1;
 10:
  write(traversed, 3: FW);
  if Pattern[PatPos] = SorText[SorPos] then begin
     write(traversed, 4: FW);
     begin
        SorPos := SorPos + 1;
        PatPos := PatPos + 1
     end
  end else begin
     write(traversed, 5: FW);
     begin
        SorPos := SorPos - PatPos + 2;
        PatPos := 1
     end
  end;
  write(traversed, 6: FW);
  if not ((PatPos > PatLen) or (SorPos > SorLen)) then
     goto 10;
  write(traversed, 7: FW);
  if PatPos > PatLen then begin
     write(traversed, 8: FW);
     StringMatch := SorPos - PatLen
  end else begin
  write(traversed, 9: FW);
     StringMatch := 0
  end;
  write(traversed, 10: FW);
  write(traversed, 11: FW)
 end; { StringMatch }

begin
```

```
  rewrite(traversed);
  writeln('ENTER THE TEXT');
  i := 1;
  while not eoln and (i <= MAX) do begin

   read(Txt[i]);
   i := i + 1

  end;
  TxtLen := i - 1;
  readln;
  writeln('ENTER THE PATTERN');
  i := 1;
  while not eoln and (i <= LENGTH) do begin

   read(Pat[i]);
   i := i + 1

  end;
  readln;
  result := StringMatch(Pat, Txt, i - 1, TxtLen);
  writeln('The pattern first appears at position ', result: 3, ' in the text.')

end. { TestStringMatch }
```

Examining the annotated flow graph, we see that in order to execute a path from 2 to 4 which is definition clear with respect to PatPos, a test case in which the first character of the string matches the first character of the pattern is needed. We run the program on such a test case, adding its trace to those produced by the test cases run previously.

```
>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: y
Command line arguments? (Y/N) [Y]
>>: n

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
The
The pattern first appears at position   1 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: n

>>>: check
```

Still need to exercise all of the following of def-clear paths:

with respect to	from	to
SorPos	5	8
PatPos	4	(7, 9)
PatPos	5	(7, 8)

To look at these again use the command 'view results'.

Examining the annotated flow graph, we see that in order to execute a path from 5 to 8 which is definition clear with respect to SorPos, a test case in which the pattern is the null string is needed. We run the program on such a test case, adding its trace to those produced by the test cases run previously.

>>>: **run**
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: **y**
Command line arguments? (Y/N) [Y]
>>: **n**

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN

The pattern first appears at position 2 in the text.
Examining the program's output, we see that the program has reported that the null string first appears in position 2 of the string. This is an error! The reader should note that this bug is guaranteed to be detected by any test set which is adequate according to the all-uses criterion. Having discovered a bug, we save the ASSET session and prepare to report the error.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: **n**

>>>: **save**

Note that to cover the one remaining association, (4,(7,9),PatPos), we would have to include a test case in which the first k characters of the pattern matched the last k characters of the text, for some k s.t. 0 < k < the length of the pattern.

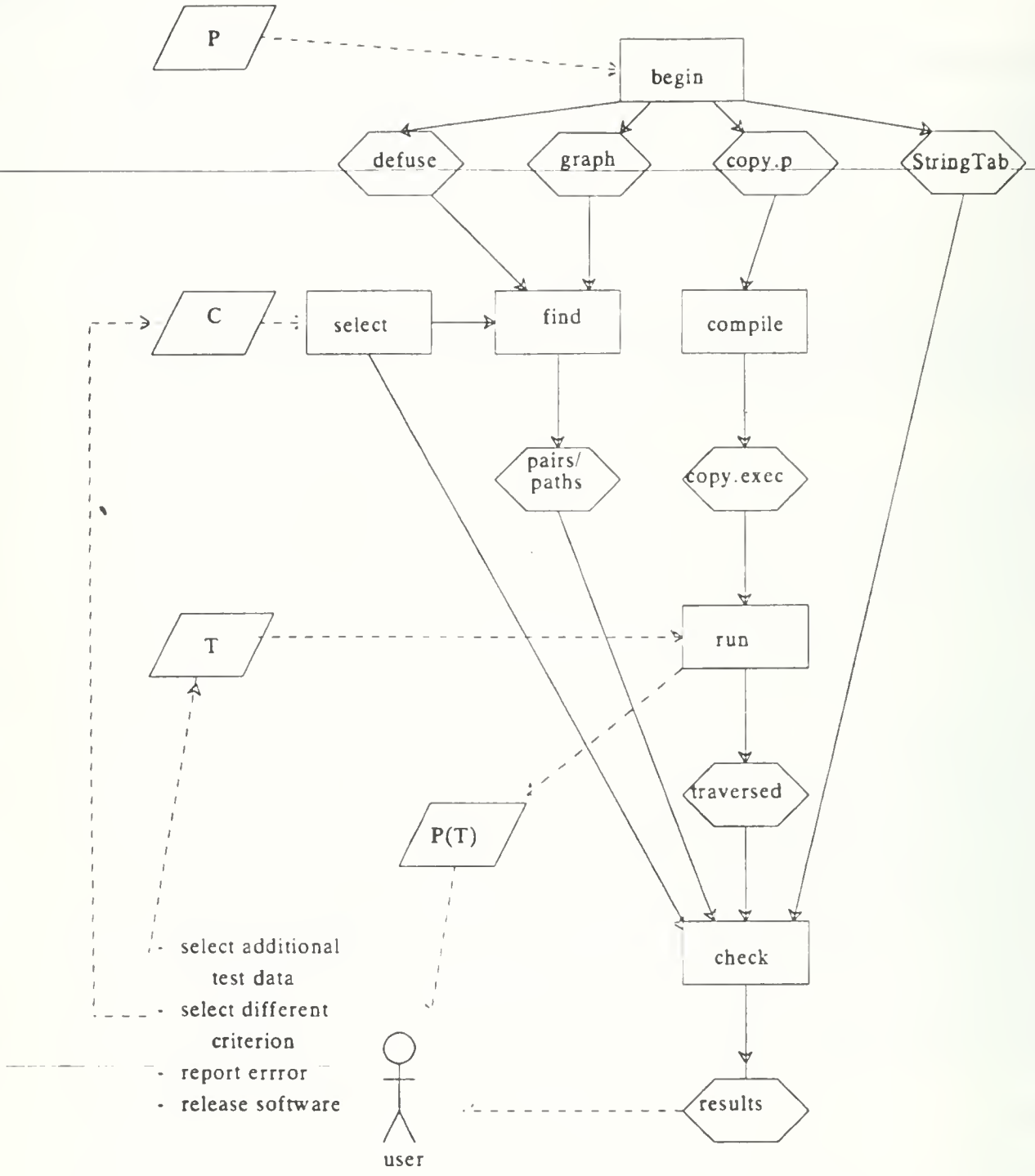


Figure 1
Schematic diagram of ASSET
P is program. C is criterion. T is Test Set.

7. REFERENCES

1. S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, Vol. SE-11, No.4, April 1985, pp. 367-375.

2. P.G. Frankl, S.N. Weiss and E.J. Weyuker, "ASSET: A System to Select and Evaluate Tests", *Proceedings of the IEEE Conference on Software Tools*, New York, April 1985.

3. P.G. Frankl and E.J. Weyuker, "A Data Flow Testing Tool," *Proceedings of IEEE Softfair II*, San Francisco, Dec. 1985.

4. P.G. Frankl and E.J. Weyuker, "Data Flow Testing in the Presence of Unexecutable Paths", *Proceedings of the IEEE Workshop on Software Testing*, Banff, Canada, July 1986.

APPENDIX I: INSTALLING ASSET

This appendix describes how to install asset. We recommend that you read these directions carefully *before* you begin the installation.

Installation

The tarfile on your tape has a directory "asset" with three subdirectories, "asset.src", "assetlib", and "asset.man" and a csh-command-script, "MakeAsset". "asset.src" contains the source code and "assetlib" contains various other files used by ASSET. "asset.src" has five subdirectories, "interface","module1","module2","module3","module4", and "module5", each of which has the source for one of ASSET's modules. The code in "interface" is written in C, and all of the other modules are written in Berkeley Pascal. After using "tar" to read the tape, perform the following steps:

1. When installing ASSET, there are some pathnames that may have to be modified in the macros of the Makefile found in the directory "asset/asset.src/interface" which contains the source code for the ASSET user interface. When this Makefile is used to make the executable file "asset", the full pathname of the directory "assetlib" is hard coded into the source file before it is compiled. Thus, the macro ASSETLIB in the Makefile must be assigned this full pathname; note that this change must be made. Also, when you make the interface, the executable file made called "asset" is left in the "interface" directory. If you want the executable file to be placed somewhere else, the DEST macro of the Makefile may be modified to be the (full) pathname of the directory where you want to install "asset"; then if you go to the directory "asset.src/interface" and execute the command

   make install

   "asset" will be installed in the correct location.

2. Change directories to "assetdir" then execute "MakeASSET". This command causes the code in each of the subdirectories of asset.src to be compiled (and therefore will take a long time to execute), then creates links from assetlib and bin to the executable code.

3. Include the directory containing the executable file *asset* in the search-path of anyone who plans to use ASSET. If you have not moved this file (according to the directions in step 1 above) this will be the directory *asset.src/interface*.